



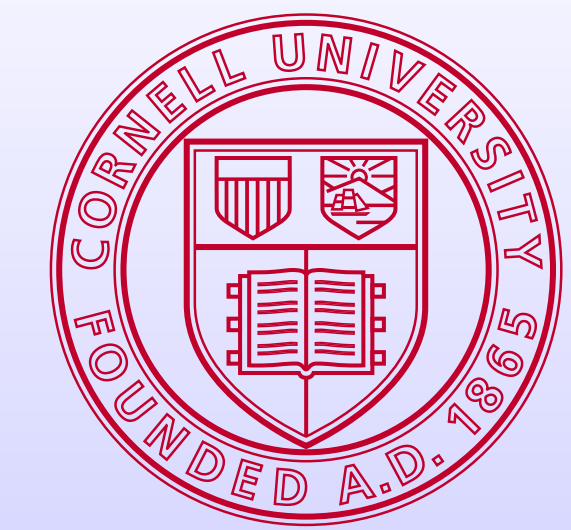
# A Cache Conflict Analysis Tool

Vincent Weaver, Cornell University

CASC, Martin Schulz

Lawrence Livermore National Laboratory

11 August 2005



## Abstract

The Cache Conflict Analysis Tool takes memory access traces from an instrumented program and runs them through a cache simulator, tracking which data structures conflict with each other. We will use the results of the tool to perform automatic cache conscious data placement, resulting in increased program performance.

## Background

- The cache location of a data structure is dependent on its location in main memory.
- The layout of data (both static and dynamic) in main memory can dramatically affect cache performance due to conflicts, and can adversely affect performance of an application.
- Compilers typically do not take cache conflict behavior into account when creating an executable, nor does the memory subsystem or operating system at time of execution.
- By analyzing cache behavior with this tool, hopefully more optimal data placement can be achieved.

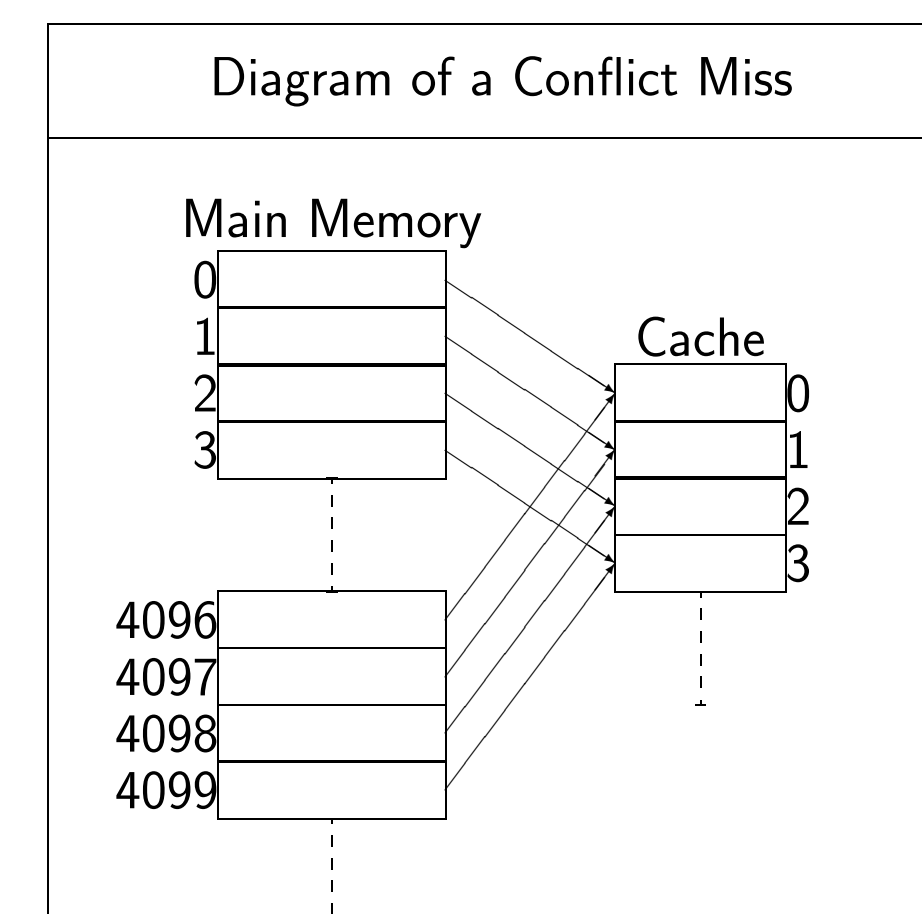
Below is an example of how in a direct mapped 4k cache with a 32 byte blocksize, every 4096th byte in main memory will map to the same area of cache. So if you have two data structures in use at the same time that are 4096 bytes apart, both will not fit in the cache at the same time and performance will be adversely affected.

### Pseudo-code of a Conflict Miss

Allocate A[512] (allocated at offset 0)  
Allocate B[512] (allocated at offset 4096)

```
Loop i=0..N
  Loop j=0..511
    A[j]=A[j]+B[j]
```

Without conflicts, A and B should be in the cache after the first access. Due to conflicts, each access causes a miss.



## Results

Below are some results from running different benchmarks through the tool, with a simulated cache configuration similar to that of a Pentium 4 (8K 4-way 64byte L1, 512K 8-way 64byte L2)

equake from the spec2k benchmarks

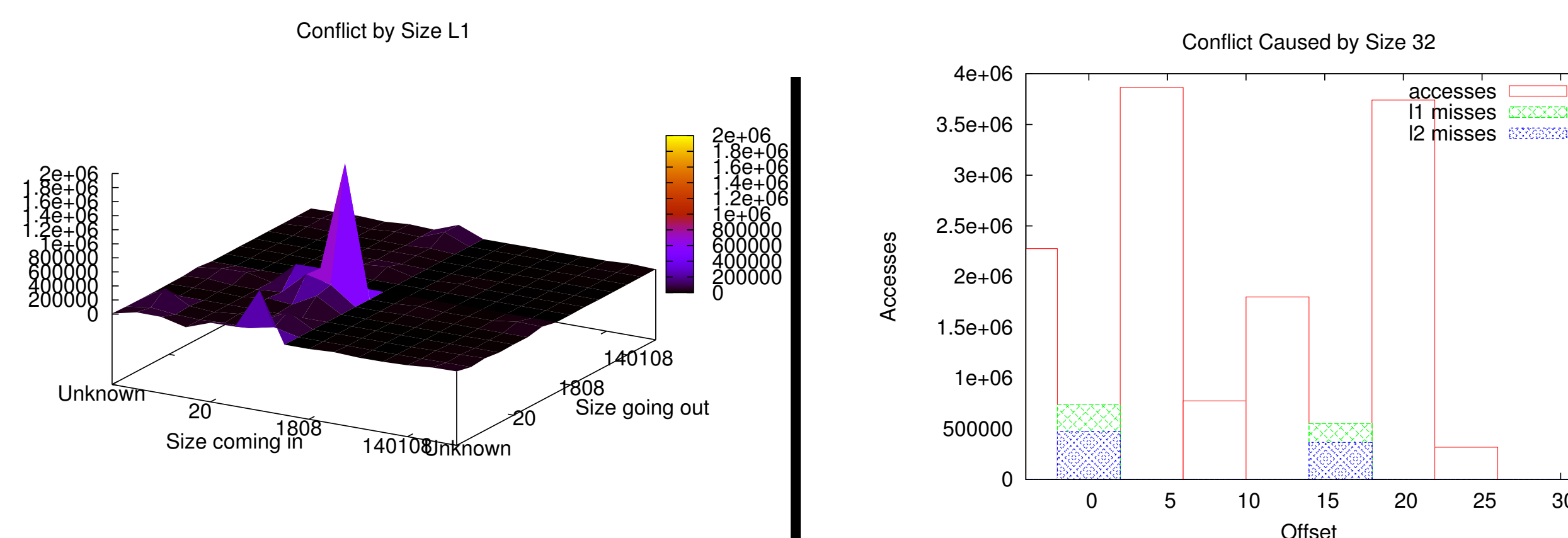


FIGURE 1: Conflicts by size in the L1 Cache. The biggest peak is that of 24-byte allocations (corresponding to an array of 3 doubles) kicking each other out of the cache.

FIGURE 2: Accesses, both total and those causing misses, for offsets in the 24-byte allocations (plus 8 bytes of padding added by the system's malloc() routine).

smg2k from the ASCI Purple Benchmarks

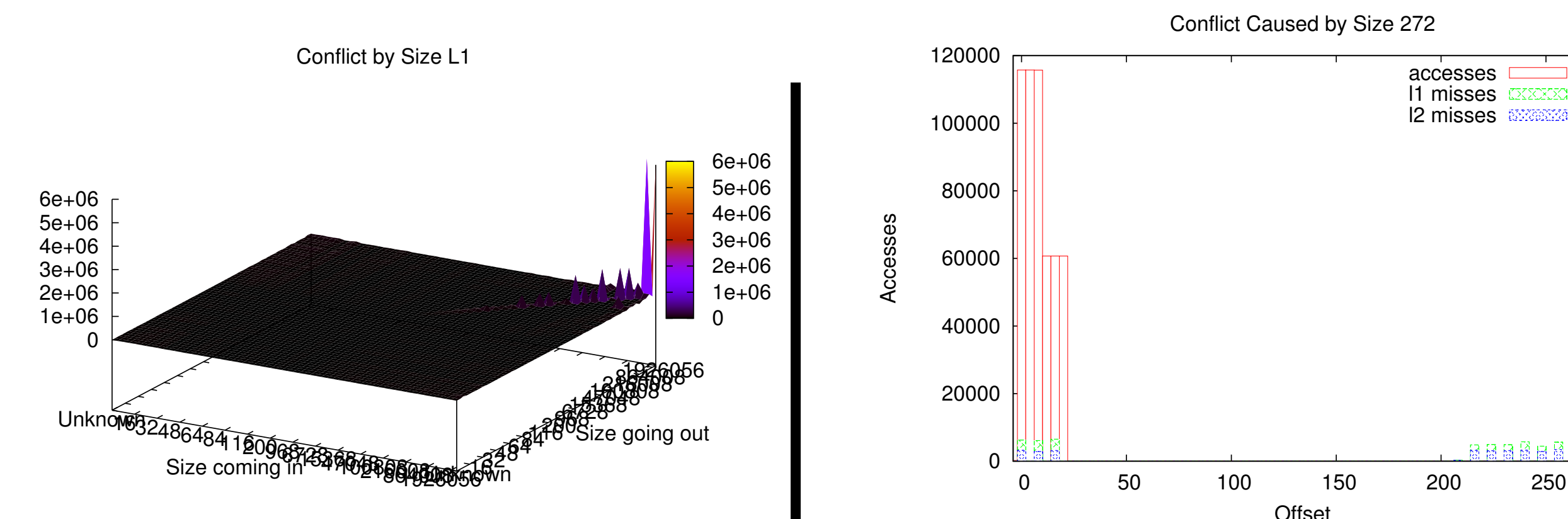
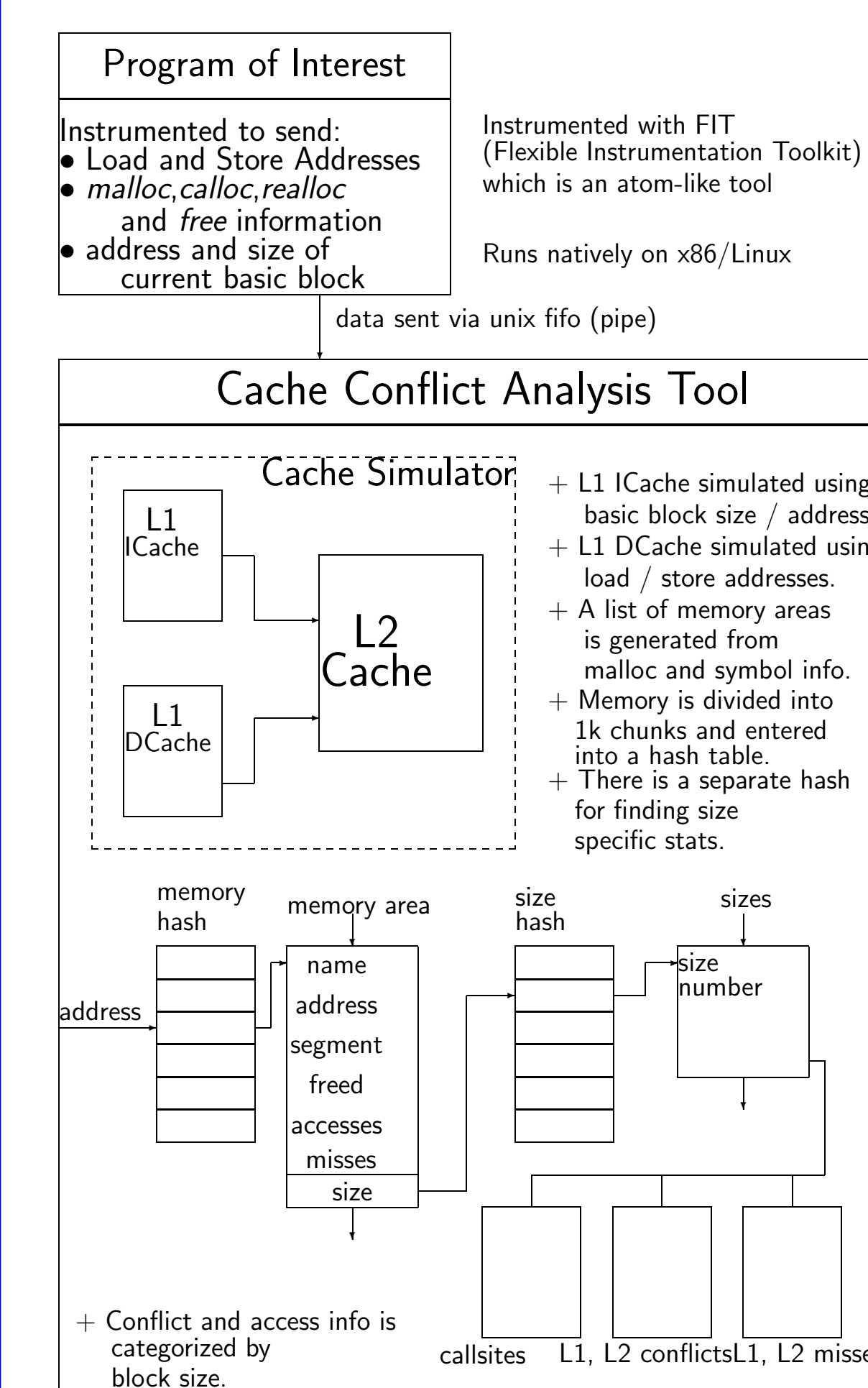


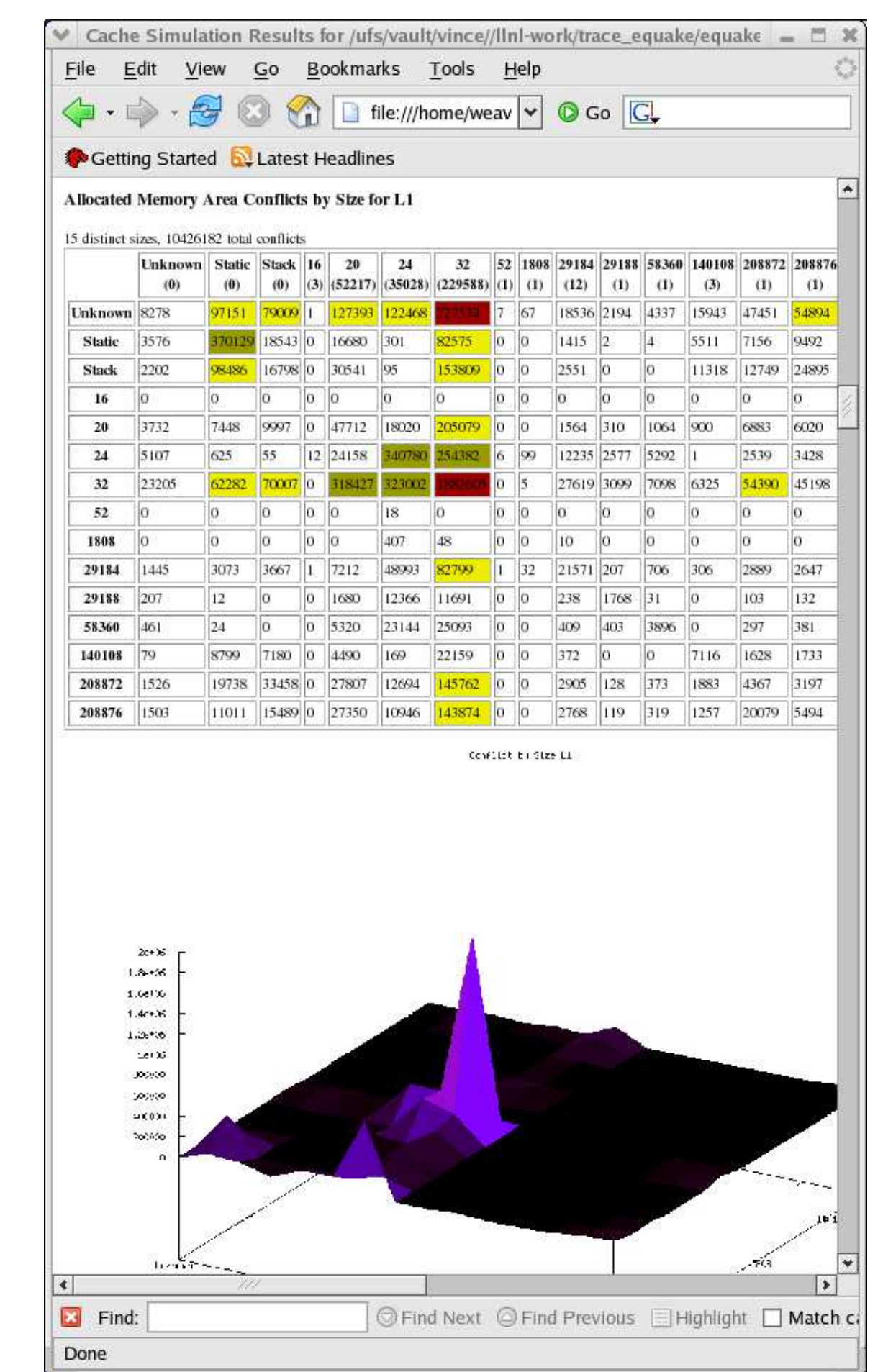
FIGURE 3: Conflicts by size in the L1 Cache. The biggest peaks here are from large allocations (bigger than the cache size) causing capacity misses.

FIGURE 4: Accesses, both total and those causing misses, for offsets in the 272-byte allocations. The large gap of unused space in the middle might open opportunities for better cache placement.

## Implementation



Memory access info is collected from the program on the fly, run through a cache simulator, and various statistics are recorded.



Results are presented as HTML with graphics.

## Future Work

- Implement automatic cache-conscious data placement:
  - + by hand through code organization
  - + automatically at runtime via customized memory management
  - + using hardware remapping mechanisms (such as IMPULSE)
- Enable instrumentation by atom or valgrind in addition to FIT.
- Instrument more benchmarks.